Problem Set 6 - Waze Shiny Dashboard

Daisy Wang

2024-11-22

1. **ps6:** Due Sat 23rd at 5:00PM Central. Worth 100 points (80 points from questions, 10 points for correct submission and 10 points for code style) + 10 extra credit.

We use (*) to indicate a problem that we think might be time consuming.

Steps to submit (10 points on PS6)

1. "This submission is my work alone and complies with the 30538 integrity policy." Add your initials to indicate your agreement: **___**
2. "I have uploaded the names of anyone I worked with on the problem set **here**" **___** (2 point)
3. Late coins used this pset: **___** Late coins left after submission: **___**
4. Before starting the problem set, make sure to read and agree to the terms of data usage for the Waze data here.
5. Knit your `ps6.qmd` as a pdf document and name it `ps6.pdf`.
6. Push your `ps6.qmd`, `ps6.pdf`, `requirements.txt`, and all created folders (we will create three Shiny apps so you will have at least three additional folders) to your Github repo (5 points). It is fine to use Github Desktop.
7. Submit `ps6.pdf` and also link your Github repo via Gradescope (5 points)
8. Tag your submission in Gradescope. For the Code Style part (10 points) please tag the whole correspondingsection for the code style rubric.

Notes: see the Quarto documentation (link) for directions on inserting images into your knitted document.

IMPORTANT: For the App portion of the PS, in case you can not arrive to the expected functional dashboard we will need to take a look at your `app.py` file. You can use the following

code chunk template to "import" and print the content of that file. Please, don't forget to also tag the corresponding code chunk as part of your submission!

```
def print_file_contents(file_path):
    """Print contents of a file."""
    try:
        with open(file_path, 'r') as f:
            content = f.read()
            print("```python")
            print(content)
            print("```")
    except FileNotFoundError:
        print("```python")
        print(f"Error: File '{file_path}' not found")
        print("```")
    except Exception as e:
        print("```python")
        print(f"Error reading file: {e}")
        print("```")

print_file_contents("./top_alerts_map_byhour/app.py") # Change accordingly
```

```
DataTransformerRegistry.enable('default')
```

Background

Data Download and Exploration (20 points)

1.

```
# Define the zip file and extraction directory
zip_file = 'waze_data.zip'
extract_dir = 'waze_data'

# Create the extraction directory if it doesn't exist
os.makedirs(extract_dir, exist_ok=True)

# Unzip the file
with zipfile.ZipFile(zip_file, 'r') as zip_ref:
    zip_ref.extractall(extract_dir)
```

```python
# Path to the sample CSV file
sample_csv_path = os.path.join(extract_dir, 'waze_data_sample.csv')

# Load the sample data into a pandas DataFrame
df_sample = pd.read_csv(sample_csv_path)

# Display the first few rows
print("First few rows of the sample data:")
print(df_sample.head())

# Exclude specified columns
columns_to_inspect = [col for col in df_sample.columns if col not in ['ts',
 ↪  'geo', 'geoWKT']]

# Function to map pandas data types to Altair data types
def map_dtype_to_altair(dtype):
    if pd.api.types.is_numeric_dtype(dtype):
        return 'Quantitative'
    elif pd.api.types.is_datetime64_any_dtype(dtype):
        return 'Temporal'
    elif pd.api.types.is_categorical_dtype(dtype) or
     ↪  pd.api.types.is_object_dtype(dtype):
        return 'Nominal'
    else:
        return 'Nominal'  # Default to Nominal if unsure

# Get data types for each column
data_types = {col: map_dtype_to_altair(df_sample[col].dtype) for col in
 ↪  columns_to_inspect}

# Print variable names and their data types
print("\nVariable names and their data types (Altair syntax):")
for var, dtype in data_types.items():
    print(f"- {var}: {dtype}")
```

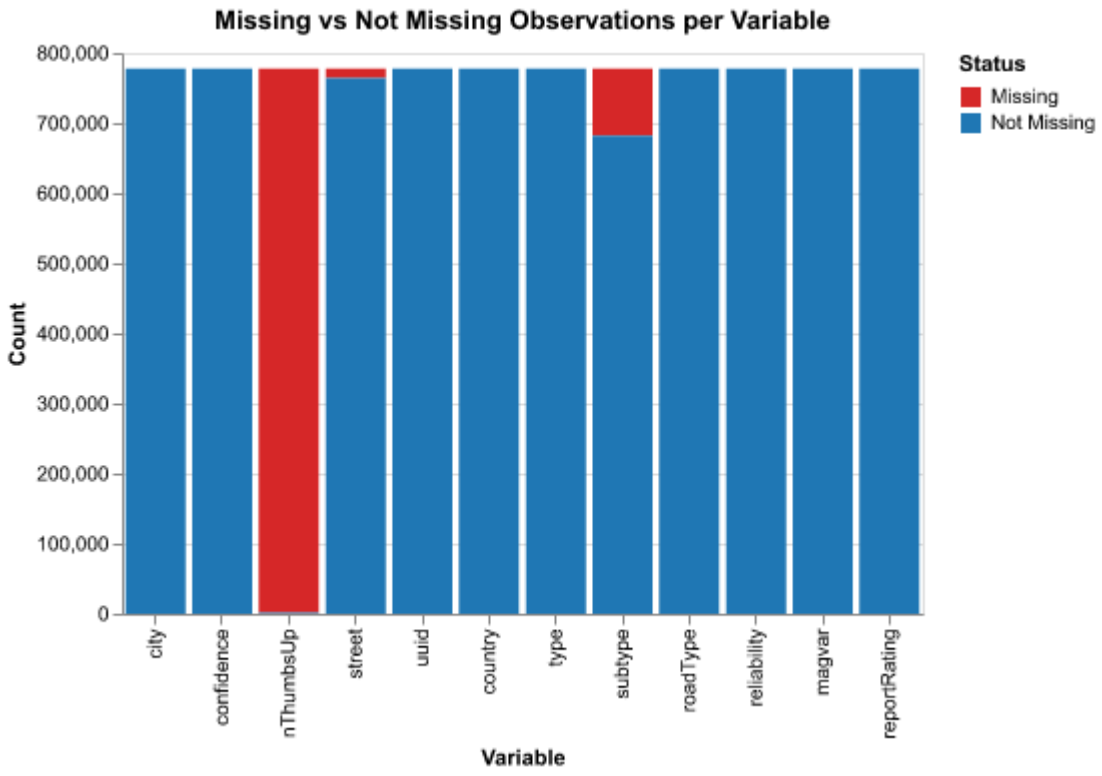
```
First few rows of the sample data:
   Unnamed: 0         city  confidence  nThumbsUp        street  \
0      584358  Chicago, IL           0        NaN           NaN
1      472915  Chicago, IL           0        NaN        I-90 E
2      550891  Chicago, IL           0        NaN        I-90 W
3      770659  Chicago, IL           0        NaN           NaN
4      381054  Chicago, IL           0        NaN  N Pulaski Rd
```

```
                                   uuid country         type  \
0  c9b88a12-79e8-44cb-aadd-a75855fc4bcb      US          JAM
1  7c634c0a-099c-4262-b57f-e893bdebce73      US  ROAD_CLOSED
2  7aa3c61a-f8dc-4fe8-bbb0-db6b9e0dc53b      US       HAZARD
3  3b95dd2f-647c-46de-b4e1-8ebc73aa9221      US       HAZARD
4  13a5e230-a28a-4bf4-b928-bc1dd38850e0      US          JAM

                         subtype  roadType  reliability  magvar  \
0                            NaN        17            5     116
1              ROAD_CLOSED_EVENT         3            6     173
2  HAZARD_ON_SHOULDER_CAR_STOPPED         3            5     308
3                 HAZARD_ON_ROAD        20            5     155
4              JAM_HEAVY_TRAFFIC         7            5     178

   reportRating                       ts                          geo  \
0             5  2024-07-02 18:27:40 UTC   POINT(-87.64577 41.892743)
1             0  2024-06-16 10:13:19 UTC  POINT(-87.646359 41.886295)
2             5  2024-05-02 19:01:47 UTC   POINT(-87.695982 41.93272)
3             2  2024-03-25 18:53:24 UTC  POINT(-87.669253 41.904497)
4             2  2024-06-03 21:17:33 UTC  POINT(-87.728322 41.978769)

                      geoWKT
0   Point(-87.64577 41.892743)
1  Point(-87.646359 41.886295)
2   Point(-87.695982 41.93272)
3  Point(-87.669253 41.904497)
4  Point(-87.728322 41.978769)

Variable names and their data types (Altair syntax):
- Unnamed: 0: Quantitative
- city: Nominal
- confidence: Quantitative
- nThumbsUp: Quantitative
- street: Nominal
- uuid: Nominal
- country: Nominal
- type: Nominal
- subtype: Nominal
- roadType: Quantitative
- reliability: Quantitative
- magvar: Quantitative
- reportRating: Quantitative
```

2.

```
# Path to the full CSV file
full_csv_path = os.path.join(extract_dir, 'waze_data.csv')

# Load the full data into a pandas DataFrame
df_full = pd.read_csv(full_csv_path)

# Exclude specified columns
columns_to_check = [col for col in df_full.columns if col not in ['ts',
 ↪  'geo', 'geoWKT']]

# Calculate missing and non-missing counts for each column
missing_counts = df_full[columns_to_check].isnull().sum()
non_missing_counts = df_full[columns_to_check].shape[0] - missing_counts

# Create a DataFrame for plotting
missing_df = pd.DataFrame({
    'Variable': columns_to_check,
    'Missing': missing_counts,
    'Not Missing': non_missing_counts
})

# Melt the DataFrame for Altair
missing_melted = missing_df.melt(id_vars='Variable', value_vars=['Missing',
 ↪  'Not Missing'],
                                 var_name='Status', value_name='Count')

# Create a stacked bar chart using Altair
stacked_bar = alt.Chart(missing_melted).mark_bar().encode(
    x=alt.X('Variable:N', sort='-y'),
    y=alt.Y('Count:Q'),
    color=alt.Color('Status:N', scale=alt.Scale(domain=['Missing', 'Not
 ↪  Missing'],
                                                range=['#d62728',
                                                 ↪  '#1f77b4'])),
    tooltip=['Variable', 'Status', 'Count']
).properties(
    title='Missing vs Not Missing Observations per Variable',
    width=400,
    height=280
).interactive()
```

```
# Display the chart
stacked_bar.display()

# Identify variables with NULL values
variables_with_null = missing_counts[missing_counts > 0].index.tolist()
print("\nVariables with NULL values:")
for var in variables_with_null:
    print(f"- {var}")

# Identify variable with highest share of missing observations
missing_share = missing_counts / df_full.shape[0]
var_highest_missing = missing_share.idxmax()
print(f"\nVariable with the highest share of missing observations:
 ↪  {var_highest_missing} "
      f"({missing_share[var_highest_missing]:.2%} missing)")
```




```
Variables with NULL values:
```

```
- nThumbsUp
- street
- subtype

Variable with the highest share of missing observations: nThumbsUp (99.82%
missing)
```

3.

```
# ------------------------------
# Step 1: Display unique types and subtypes
# ------------------------------
unique_types = df_full['type'].dropna().unique()
print("Unique Types:")
for t in unique_types:
    print(f"- {t}")

unique_subtypes = df_full['subtype'].dropna().unique()
print("\nUnique Subtypes:")
for st in unique_subtypes:
    print(f"- {st}")

# ------------------------------
# Step 2: Create a copy to avoid modifying the original dataframe
# ------------------------------
df_cleaned = df_full.copy()

# ------------------------------
# Step 3: Replace Underscores with Spaces and Apply Title Case
# ------------------------------

# Replace underscores with spaces and apply title case
df_cleaned['type'] = df_cleaned['type'].str.replace('_', ' ').str.title()
df_cleaned['subtype'] = df_cleaned['subtype'].str.replace('_', '
↪  ').str.title()

# ------------------------------
# Step 4: Rename Specific Subtypes to Enhance Clarity
# ------------------------------

# Define a mapping for specific subtypes to rename them
rename_mapping = {
    'Road Closed Hazard': 'Closure Due to Hazard'
```

```
}

# Apply the renaming
df_cleaned['subtype'] = df_cleaned['subtype'].replace(rename_mapping)

# ------------------------------
# Step 5: Identify Types Suitable for Sub-subtypes
# ------------------------------

# Calculate the number of unique subtypes per type
subtype_counts =
 ↪  df_cleaned.groupby('type')['subtype'].nunique().reset_index()
subtype_counts = subtype_counts.rename(columns={'subtype':
 ↪  'unique_subtypes'})
print("\nNumber of Unique Subtypes per Type:")
print(subtype_counts)

# Identify types with more than one subtype
types_with_multiple_subtypes =
 ↪  subtype_counts[subtype_counts['unique_subtypes'] > 1]['type'].tolist()
print(f"\nTypes with multiple subtypes (eligible for sub-subtypes):
 ↪  {len(types_with_multiple_subtypes)}")
for t in types_with_multiple_subtypes:
    print(f"- {t}")

# ------------------------------
# Step 6: Create a Clean, Readable Hierarchy Without "Other" Category
# ------------------------------

# Initialize hierarchy dictionary
hierarchy = {}

for t in types_with_multiple_subtypes:
    # Extract subtypes for the current type, excluding NA
    subtypes = df_cleaned[df_cleaned['type'] ==
 ↪  t]['subtype'].dropna().unique()

    # Initialize subcategory structure
    if t == "Hazard":
        # Further categorize 'Hazard' into sub-subtypes
        hazard_subcategories = {
            "On Road": [],
```

```
            "On Shoulder": [],
            "Weather": []
        }
        for st in subtypes:
            if st.startswith("Hazard On Road"):
                # Remove 'Hazard On Road' prefix
                sub_detail = st.replace("Hazard On Road", "").strip()
                if sub_detail:
                    hazard_subcategories["On Road"].append(sub_detail)
            elif st.startswith("Hazard On Shoulder"):
                # Remove 'Hazard On Shoulder' prefix
                sub_detail = st.replace("Hazard On Shoulder", "").strip()
                if sub_detail:
                    hazard_subcategories["On Shoulder"].append(sub_detail)
            elif st.startswith("Hazard Weather"):
                # Remove 'Hazard Weather' prefix
                sub_detail = st.replace("Hazard Weather", "").strip()
                if sub_detail:
                    hazard_subcategories["Weather"].append(sub_detail)
            else:
                # Ignore any subtypes that don't fit into predefined
                 ↪ subcategories
                pass

        # Remove empty subcategories
        hazard_subcategories = {k: v for k, v in hazard_subcategories.items()
↪   if v}

        hierarchy[t] = hazard_subcategories
    else:
        # For other types, simply assign the cleaned subtypes
        # Remove the type name from the subtype if present
        cleaned_subtypes = [s.replace(f"{t} ", "").strip() for s in subtypes]
        # Replace multiple spaces with single space
        cleaned_subtypes = [' '.join(sub.split()) for sub in
↪   cleaned_subtypes]
        hierarchy[t] = list(cleaned_subtypes)

# -----------------------------
# Step 7: Handle "Unclassified" Subtypes
# -----------------------------
```

```
# Identify types with NA subtypes
types_with_na_subtypes =
 ↪  df_cleaned[df_cleaned['subtype'].isnull()]['type'].unique()
num_types_with_na_subtypes = len(types_with_na_subtypes)
print(f"\nThere are {num_types_with_na_subtypes} Type(s) with NA Subtypes:")
for t in types_with_na_subtypes:
    print(f"- {t}")

# Decide to keep NA subtypes and rename them as 'Unclassified'
keep_na_subtypes = True  # Set to False if you decide to drop NA subtypes

if keep_na_subtypes and num_types_with_na_subtypes > 0:
    for t in types_with_na_subtypes:
        type_clean = t.replace("_", " ").title()
        if type_clean == "Hazard":
            # Add 'Unclassified' to each subcategory
            for subcategory in hierarchy[type_clean]:
                hierarchy[type_clean][subcategory].append("Unclassified")
        else:
            # For other types, append 'Unclassified' directly to the subtypes
             ↪  list
            hierarchy[type_clean].append("Unclassified")

# -----------------------------
# Step 8: Remove "Hazard" Prefix from Subsubtypes
# -----------------------------

# Since we've already removed the "Hazard On Road", "Hazard On Shoulder", and
 ↪  "Hazard Weather" prefixes,
# there's no need for further removal. However, to ensure no residual
 ↪  "Hazard" exists, we can perform a check.

for t in hierarchy:
    if t == "Hazard":
        for subcategory in hierarchy[t]:
            # Remove any residual "Hazard" in subsubtypes
            hierarchy[t][subcategory] = [sub.replace("Hazard ", "").strip()
 ↪  for sub in hierarchy[t][subcategory]]

# -----------------------------
# Step 9: Final Hierarchy Display
# -----------------------------
```

```
def print_hierarchy(hierarchy_dict, indent=0):
    for type_name, sub_info in hierarchy_dict.items():
        print(f"{'  ' * indent}- {type_name}")
        if isinstance(sub_info, dict):
            for subcategory, sub_list in sub_info.items():
                print(f"{'  ' * (indent + 1)}- {subcategory}")
                for sub in sub_list:
                    print(f"{'  ' * (indent + 2)}- {sub}")
        elif isinstance(sub_info, list):
            for sub in sub_info:
                print(f"{'  ' * (indent + 1)}- {sub}")
        else:
            print(f"{'  ' * (indent + 1)}- {sub_info}")

print("\nHierarchy of Types and Subtypes:")
print_hierarchy(hierarchy)
```

```
Unique Types:
- JAM
- ACCIDENT
- ROAD_CLOSED
- HAZARD

Unique Subtypes:
- ACCIDENT_MAJOR
- ACCIDENT_MINOR
- HAZARD_ON_ROAD
- HAZARD_ON_ROAD_CAR_STOPPED
- HAZARD_ON_ROAD_CONSTRUCTION
- HAZARD_ON_ROAD_EMERGENCY_VEHICLE
- HAZARD_ON_ROAD_ICE
- HAZARD_ON_ROAD_OBJECT
- HAZARD_ON_ROAD_POT_HOLE
- HAZARD_ON_ROAD_TRAFFIC_LIGHT_FAULT
- HAZARD_ON_SHOULDER
- HAZARD_ON_SHOULDER_CAR_STOPPED
- HAZARD_WEATHER
- HAZARD_WEATHER_FLOOD
- JAM_HEAVY_TRAFFIC
- JAM_MODERATE_TRAFFIC
- JAM_STAND_STILL_TRAFFIC
```

```
- ROAD_CLOSED_EVENT
- HAZARD_ON_ROAD_LANE_CLOSED
- HAZARD_WEATHER_FOG
- ROAD_CLOSED_CONSTRUCTION
- HAZARD_ON_ROAD_ROAD_KILL
- HAZARD_ON_SHOULDER_ANIMALS
- HAZARD_ON_SHOULDER_MISSING_SIGN
- JAM_LIGHT_TRAFFIC
- HAZARD_WEATHER_HEAVY_SNOW
- ROAD_CLOSED_HAZARD
- HAZARD_WEATHER_HAIL

Number of Unique Subtypes per Type:
          type  unique_subtypes
0     Accident                2
1       Hazard               19
2          Jam                4
3  Road Closed                3

Types with multiple subtypes (eligible for sub-subtypes): 4
- Accident
- Hazard
- Jam
- Road Closed

There are 4 Type(s) with NA Subtypes:
- Jam
- Accident
- Road Closed
- Hazard

Hierarchy of Types and Subtypes:
- Accident
  - Major
  - Minor
  - Unclassified
- Hazard
  - On Road
    - Car Stopped
    - Construction
    - Emergency Vehicle
    - Ice
    - Object
```

```
    - Pot Hole
    - Traffic Light Fault
    - Lane Closed
    - Road Kill
    - Unclassified
  - On Shoulder
    - Car Stopped
    - Animals
    - Missing Sign
    - Unclassified
  - Weather
    - Flood
    - Fog
    - Heavy Snow
    - Hail
    - Unclassified
- Jam
  - Heavy Traffic
  - Moderate Traffic
  - Stand Still Traffic
  - Light Traffic
  - Unclassified
- Road Closed
  - Event
  - Construction
  - Closure Due to Hazard
  - Unclassified
```

The HAZARD type subtypes can be organized into the following hierarchy: - **On Road** - Car Stopped - Construction - Emergency Vehicle - Ice - Object - Pot Hole - Traffic Light Fault - Lane Closed - Road Kill - **On Shoulder** - Car Stopped - Animals - Missing Sign - **Weather** - Flood - Fog - Heavy Snow - Hail

I think we should keep the NA subtypes for the following reasons: 1. Removing NA subtypes may lead to the loss of potentially valuable information. Even if a subtype is unknown or unclassified, the main type (HAZARD, ACCIDENT, etc.) still provides context. 2. Keeping NA subtypes, labeled as "Unclassified", ensures that all observations are accounted for. This avoids creating gaps in data, making it easier to analyze and interpret. 3. Unclassified entries might provide valuable insights later. For instance, if additional data becomes available, these entries can be reclassified appropriately.

4.
5.

```
# Step a.1: Extract unique 'type' and 'subtype' combinations from the
 ↪  original dataset
unique_combinations = df_full[['type',
 ↪  'subtype']].drop_duplicates().reset_index(drop=True)

# Step a.2: Create the crosswalk DataFrame by copying the unique combinations
crosswalk_df = unique_combinations.copy()

# Step a.3: Add three new columns initialized as None (you can use empty
 ↪  strings '' if preferred)
crosswalk_df['updated_type'] = None
crosswalk_df['updated_subtype'] = None
crosswalk_df['updated_subsubtype'] = None
```

2.

```
def map_hierarchy(row, hierarchy):
    """
    Maps the original type and subtype to updated_type, updated_subtype, and
 ↪  updated_subsubtype
    based on the provided hierarchy.
    """
    original_type = row['type']
    original_subtype = row['subtype']

    # Clean 'type' to match hierarchy keys
    type_clean = original_type.replace('_', ' ').title()

    # Handle 'subtype'
    if pd.isna(original_subtype):
        subtype_clean = 'Unclassified'
    else:
        subtype_clean = original_subtype.replace('_', ' ').title()

    # Initialize updated fields
    updated_type = None
    updated_subtype = None
    updated_subsubtype = None

    # Handle 'Unclassified' subtypes
    if subtype_clean == 'Unclassified':
        updated_type = type_clean
```

```
        updated_subtype = 'Unclassified'
        updated_subsubtype = None
        return pd.Series([updated_type, updated_subtype, updated_subsubtype])

    # Check if type exists in hierarchy
    if type_clean in hierarchy:
        sub_hierarchy = hierarchy[type_clean]

        if isinstance(sub_hierarchy, dict):
            # For types with subsubtypes (e.g., Hazard)
            # Remove the type name from subtype_clean to get the subcategory
            # Example: 'Hazard On Road' -> 'On Road'
            subtype_clean_no_type = subtype_clean.replace(type_clean,
↪   '').strip()

            # Check if subtype_clean_no_type matches any subcategory
            if subtype_clean_no_type in sub_hierarchy:
                updated_type = type_clean
                updated_subtype = subtype_clean_no_type
                updated_subsubtype = None
                return pd.Series([updated_type, updated_subtype,
                 ↪  updated_subsubtype])

            # Iterate through subcategories to find matching subsubtype
            matched = False
            for subcategory, subsubtypes in sub_hierarchy.items():
                if subtype_clean_no_type.replace(subcategory, "").strip() in
                 ↪  subsubtypes:
                    updated_type = type_clean
                    updated_subtype = subcategory.replace("On", "").strip()
                    updated_subsubtype =
↪   subtype_clean_no_type.replace(subcategory, "").strip()
                    matched = True
                    break
            if not matched:
                # If no match found in subsubtypes, assign 'Unclassified'
                updated_type = type_clean
                updated_subtype = 'Unclassified'
                updated_subsubtype = None
        elif isinstance(sub_hierarchy, list):
            # For types without subsubtypes (e.g., Jam, Accident, Road
             ↪  Closed)
```

```
            # Remove the type name from subtype_clean to get the actual
             ↪ subtype
            subtype_clean_no_type = subtype_clean.replace(type_clean,
 ↪ '').strip()

            if subtype_clean_no_type in sub_hierarchy:
                updated_type = type_clean
                updated_subtype = subtype_clean_no_type
                updated_subsubtype = None
            else:
                # If subtype not found in hierarchy list, assign
                 ↪ 'Unclassified'
                updated_type = type_clean
                updated_subtype = 'Unclassified'
                updated_subsubtype = None
    else:
        # If type not found in hierarchy, assign 'Unclassified'
        updated_type = 'Unclassified'
        updated_subtype = 'Unclassified'
        updated_subsubtype = None

    return pd.Series([updated_type, updated_subtype, updated_subsubtype])

# Apply the revised mapping function to each row in crosswalk_df
crosswalk_df[['updated_type', 'updated_subtype', 'updated_subsubtype']] =
 ↪ crosswalk_df.apply(
    map_hierarchy,
    axis=1,
    hierarchy=hierarchy
)

# Verify that the crosswalk has exactly 32 observations
print(f"\nTotal number of observations in crosswalk:
 ↪ {crosswalk_df.shape[0]}")
```

```
Total number of observations in crosswalk: 32
```

3.

```
# Merge the original data with the crosswalk on 'type' and 'subtype'
merged_df = pd.merge(
```

```
    df_full,
    crosswalk_df,
    how='left',
    on=['type', 'subtype']
)

# Filter rows where updated_type is 'Accident' and updated_subtype is
 ↪  'Unclassified'
accident_unclassified = merged_df[
    (merged_df['updated_type'] == 'Accident') &
    (merged_df['updated_subtype'] == 'Unclassified')
]

# Count the number of such rows
num_accident_unclassified = accident_unclassified.shape[0]

print(f"\nNumber of rows for Accident - Unclassified:
 ↪  {num_accident_unclassified}")
```

```
Number of rows for Accident - Unclassified: 24359
```

4.

```
# Extract unique type-subtype combinations from crosswalk_df
crosswalk_unique = crosswalk_df[['type',
 ↪  'subtype']].drop_duplicates().reset_index(drop=True)

# Extract unique type-subtype combinations from merged_df
merged_unique = merged_df[['type',
 ↪  'subtype']].drop_duplicates().reset_index(drop=True)

# Check for Missing Combinations in merged_df
missing_in_merged = pd.merge(
    crosswalk_unique,
    merged_unique,
    on=['type', 'subtype'],
    how='left',
    indicator=True
).query('_merge == "left_only"')

# Check for Extra Combinations in merged_df
```

```
extra_in_crosswalk = pd.merge(
    merged_unique,
    crosswalk_unique,
    on=['type', 'subtype'],
    how='left',
    indicator=True
).query('_merge == "left_only"')

# Step 5: Display the Results
print("\n--- Verification Results ---\n")

# a. Missing Combinations
if missing_in_merged.empty:
    print(" All crosswalk type-subtype combinations are present in the merged
     ↪ data.")
else:
    print(" Some crosswalk type-subtype combinations are missing in the
     ↪ merged data:")
    print(missing_in_merged[['type', 'subtype']])

# b. Extra Combinations
if extra_in_crosswalk.empty:
    print("\n No extra type-subtype combinations found in the merged data.")
else:
    print("\n Some type-subtype combinations in the merged data are not
     ↪ present in the crosswalk:")
    print(extra_in_crosswalk[['type', 'subtype']])
```

```
--- Verification Results ---

 All crosswalk type-subtype combinations are present in the merged data.

 No extra type-subtype combinations found in the merged data.
```

App #1: Top Location by Alert Type Dashboard (30 points)

1.

a.

```
def extract_coordinates(geo_str):
    """
    Extracts longitude and latitude from a WKT POINT string.

    Parameters:
        geo_str (str): The WKT POINT string.

    Returns:
        tuple: (longitude, latitude)
    """
    pattern = r'POINT\((-?\d+\.\d+)\s(-?\d+\.\d+)\)'
    match = re.match(pattern, geo_str)
    if match:
        lon, lat = match.groups()
        return float(lon), float(lat)
    else:
        return None, None

merged_df[['longitude', 'latitude']] = df_full['geo'].apply(
    lambda x: pd.Series(extract_coordinates(x))
)
```

b.

```
def bin_coordinate(coord):
    """
    Bins a coordinate to the nearest 0.01 by flooring.

    Parameters:
        coord (float): The coordinate value.

    Returns:
        float: Binned coordinate.
    """
    return round(coord - (coord % 0.01), 2)

# Apply binning
merged_df['binned_longitude'] = merged_df['longitude'].apply(bin_coordinate)
merged_df['binned_latitude'] = merged_df['latitude'].apply(bin_coordinate)

# Group by binned latitude and longitude and count observations
bin_counts = merged_df.groupby(['binned_latitude',
 ↪  'binned_longitude']).size().reset_index(name='counts')
```

```
# Identify the combination with the highest number of observations
top_bin = bin_counts.loc[bin_counts['counts'].idxmax()]

print(f"\nBinned Latitude-Longitude with the highest observations:
 ↪  ({top_bin['binned_latitude']}, {top_bin['binned_longitude']}) with
 ↪  {int(top_bin['counts'])} observations")
```

```
Binned Latitude-Longitude with the highest observations: (41.96, -87.75) with
26537 observations
```

c. The level of aggregation would be types and subtypes of alert.

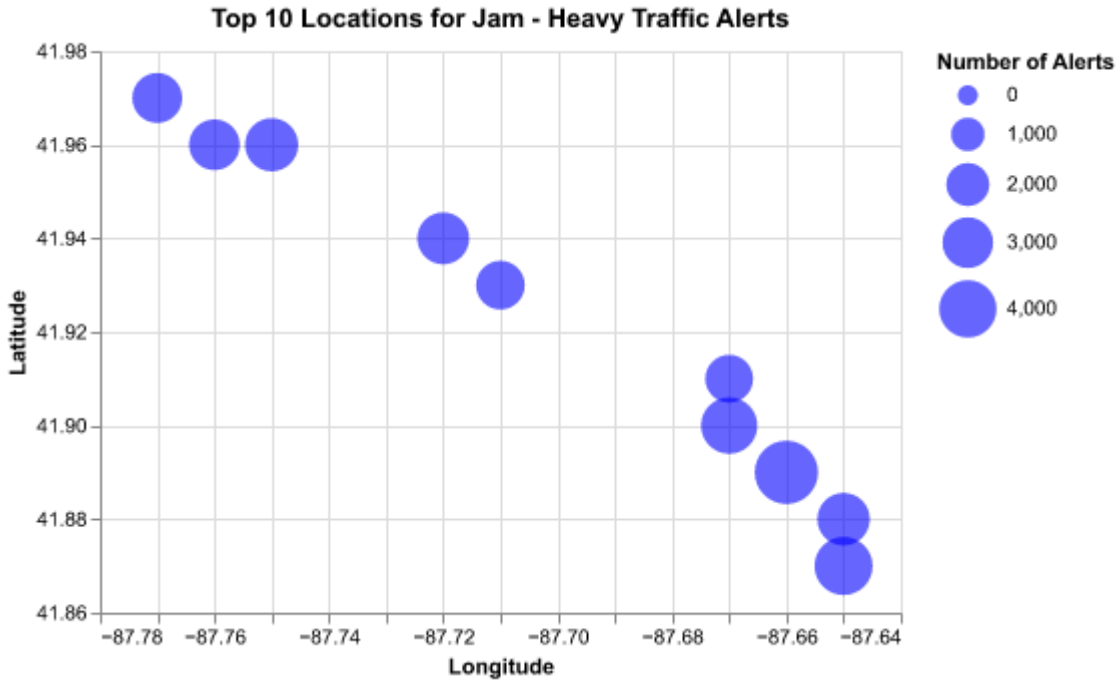
```
# Group by binned coordinates and count alerts
alert_counts = merged_df.groupby(['updated_type', 'updated_subtype',
 ↪  'binned_latitude',
 ↪  'binned_longitude']).size().reset_index(name='alert_counts')

print(f"There are {len(alert_counts)} rows of this DataFrame.")

# Save the result to a CSV file
alert_counts.to_csv('./top_alerts_map/alert_counts.csv', index=False)
```

```
There are 7455 rows of this DataFrame.
```

2.

```
# Define the chosen type and subtype
chosen_type = 'Jam'
chosen_subtype = 'Heavy Traffic'

# Filter the DataFrame
filtered_df = merged_df[
    (merged_df['updated_type'] == chosen_type) &
    (merged_df['updated_subtype'] == chosen_subtype)
]

# Group by binned coordinates and count alerts
top_bins = filtered_df.groupby(['binned_latitude',
 ↪  'binned_longitude']).size().reset_index(name='alert_counts')
```

```
# Sort the bins by alert_counts in descending order and select top 10
top_alerts = top_bins.sort_values(by='alert_counts',
 ↪  ascending=False).head(10)

# Create the scatter plot
scatter_plot = alt.Chart(top_alerts).mark_circle(color='blue',
 ↪  opacity=0.6).encode(
    x=alt.X('binned_longitude:Q',
            title='Longitude',
            scale=alt.Scale(domain=[top_alerts['binned_longitude'].min() -
 ↪  0.01,
                                    top_alerts['binned_longitude'].max() +
 ↪  0.01])),
    y=alt.Y('binned_latitude:Q',
            title='Latitude',
            scale=alt.Scale(domain=[top_alerts['binned_latitude'].min() -
 ↪  0.01,
                                    top_alerts['binned_latitude'].max() +
 ↪  0.01])),
    size=alt.Size('alert_counts:Q',
                  title='Number of Alerts',
                  scale=alt.Scale(range=[100, 1000])),
    tooltip=['binned_latitude', 'binned_longitude', 'alert_counts']
).properties(
    width=400,
    height=280,
    title=f"Top 10 Locations for {chosen_type} - {chosen_subtype} Alerts"
).interactive()

# Display the plot
scatter_plot.display()
```



3.

a.

```
# Define the URL for the Chicago Neighborhood Boundaries GeoJSON
geojson_url =
 ↪  'https://data.cityofchicago.org/api/geospatial/bbvz-uum9?method=export&format=GeoJSON'

# Define the local file path
local_geojson_path = './top_alerts_map/chicago-boundaries.geojson'

# Send a GET request to download the GeoJSON
response = requests.get(geojson_url)

# Check if the request was successful
if response.status_code == 200:
    # Ensure the directory exists
    os.makedirs(os.path.dirname(local_geojson_path), exist_ok=True)

    # Write the content to the file
    with open(local_geojson_path, 'wb') as f:
        f.write(response.content)
    print(f"GeoJSON file downloaded successfully and saved to
     ↪  {local_geojson_path}")
```

```
else:
    print(f"Failed to download GeoJSON file. Status code:
     ↪ {response.status_code}")
```

```
GeoJSON file downloaded successfully and saved to
./top_alerts_map/chicago-boundaries.geojson
```

b.

```
# Define the file path
file_path = "./top_alerts_map/chicago-boundaries.geojson"

# Load the GeoJSON data
with open(file_path) as f:
    chicago_geojson = json.load(f)

# Prepare GeoJSON for Altair
geo_data = alt.Data(values=chicago_geojson["features"])
```

4.

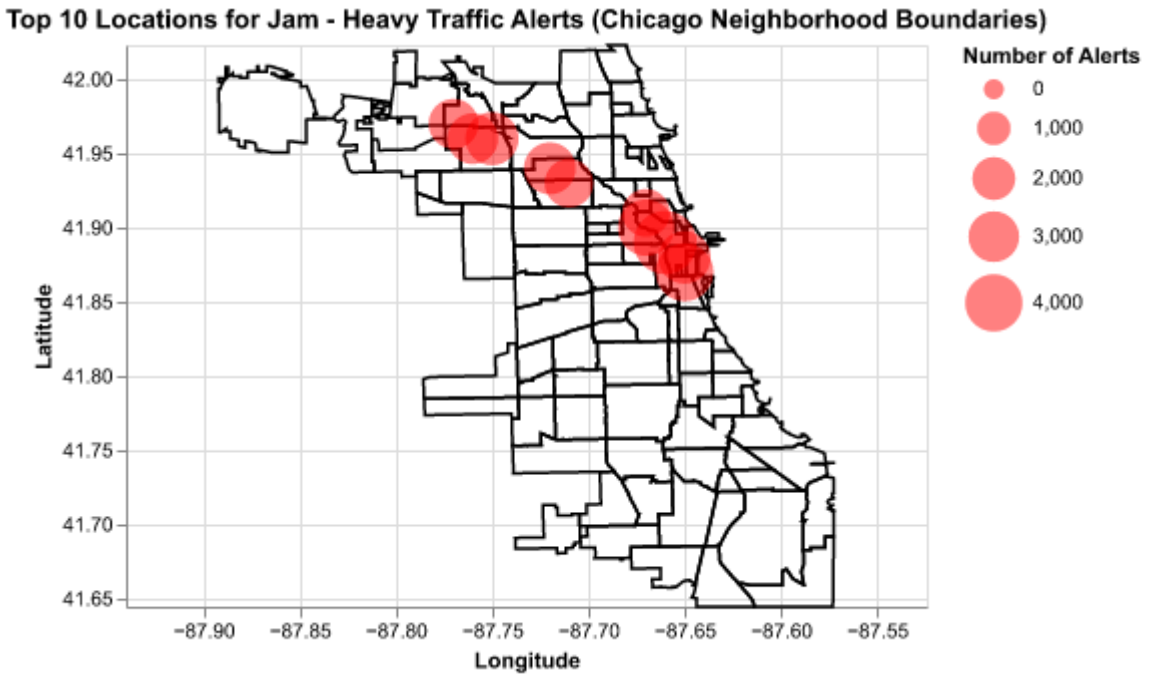
```
# Extract coordinates from GeoJSON
coordinates = [
    coord
    for feature in chicago_geojson["features"]
    for polygon in feature["geometry"]["coordinates"]
    for coord in polygon[0]  # Assuming MultiPolygon geometries
]

# Convert to a NumPy array for easier processing
coordinates = np.array(coordinates)

# Find longitude and latitude ranges
geo_longitude_range = [coordinates[:, 0].min(), coordinates[:, 0].max()]
geo_latitude_range = [coordinates[:, 1].min(), coordinates[:, 1].max()]

print(f"GeoJSON Longitude Range: {geo_longitude_range}")
print(f"GeoJSON Latitude Range: {geo_latitude_range}")

# Check longitude and latitude ranges for top_alerts
alerts_longitude_range = [top_alerts['binned_longitude'].min(),
 ↪ top_alerts['binned_longitude'].max()]
```

```
alerts_latitude_range = [top_alerts['binned_latitude'].min(),
 ↪  top_alerts['binned_latitude'].max()]

print(f"Top Alerts Longitude Range: {alerts_longitude_range}")
print(f"Top Alerts Latitude Range: {alerts_latitude_range}")

# Dynamically set the limits based on the data
longitude_limits = [
    min(geo_longitude_range[0], alerts_longitude_range[0]),
    max(geo_longitude_range[1], alerts_longitude_range[1])
]

latitude_limits = [
    min(geo_latitude_range[0], alerts_latitude_range[0]),
    max(geo_latitude_range[1], alerts_latitude_range[1])
]

print(f"Final Longitude Limits: {longitude_limits}")
print(f"Final Latitude Limits: {latitude_limits}")
```

```
GeoJSON Longitude Range: [np.float64(-87.94011408251845),
np.float64(-87.52413710388589)]
GeoJSON Latitude Range: [np.float64(41.64454312227481),
np.float64(42.02303858698943)]
Top Alerts Longitude Range: [np.float64(-87.77), np.float64(-87.65)]
Top Alerts Latitude Range: [np.float64(41.87), np.float64(41.97)]
Final Longitude Limits: [np.float64(-87.94011408251845),
np.float64(-87.52413710388589)]
Final Latitude Limits: [np.float64(41.64454312227481),
np.float64(42.02303858698943)]
```

```
# Create the base map with Chicago neighborhood boundaries
base_map = alt.Chart(geo_data).mark_geoshape(
    fill=None,  # Transparent fill
    stroke='black',  # Boundary lines
    strokeWidth=1.5  # Thickness of boundary lines
).encode(
    tooltip=[alt.Tooltip('properties.NAME:N', title='Neighborhood')]  #
 ↪  Neighborhood tooltip
).properties(
    width=400,
    height=280
```

```
).project(
    type='equirectangular'  # Use equirectangular projection for geographic
     ↪ alignment
)

# Create the scatter plot
scatter_plot = alt.Chart(top_alerts.head(10)).mark_circle(
    color='red',
    opacity=0.5
).encode(
    x=alt.X('binned_longitude:Q',
            title='Longitude',
            scale=alt.Scale(domain=longitude_limits)),  # Dynamically set
 ↪ longitude limits
    y=alt.Y('binned_latitude:Q',
            title='Latitude',
            scale=alt.Scale(domain=latitude_limits)),  # Dynamically set
 ↪ latitude limits
    size=alt.Size('alert_counts:Q',
                  title='Number of Alerts',
                  scale=alt.Scale(range=[100, 1000])),
    tooltip=[
        alt.Tooltip('binned_latitude:Q', title='Latitude'),
        alt.Tooltip('binned_longitude:Q', title='Longitude'),
        alt.Tooltip('alert_counts:Q', title='Number of Alerts')
    ]
).properties(
    width=400,
    height=280
)

# Layer the scatter plot on top of the base map
layered_map = (base_map + scatter_plot).properties(
    title=f"Top 10 Locations for {chosen_type} - {chosen_subtype} Alerts
 ↪ (Chicago Neighborhood Boundaries)"
)

# Display the layered map
layered_map.display()
```



5.

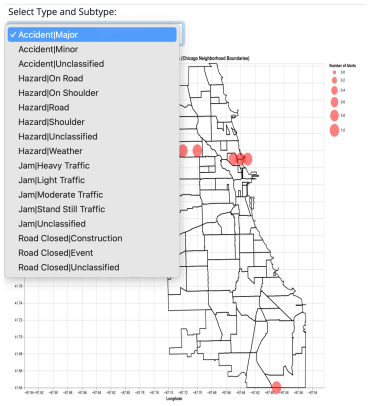


Figure 1: Dropdown Menu

a.

There are in total 17 type x subtype combinations in my dropdown menu.

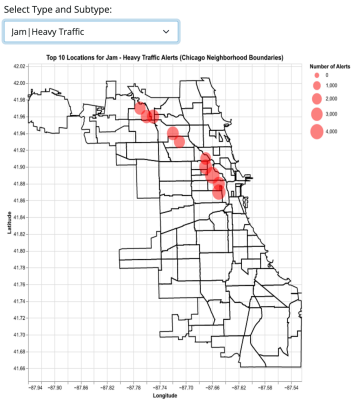


Figure 2: Jam Heavy Traffic Alerts

b.

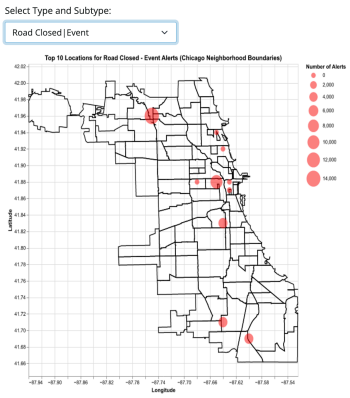


Figure 3: Road Closed Event Alerts

c.

The map shows that the most frequent locations for 'Road Closed - Event' alerts are concentrated in specific areas, as represented by the size of the red circles. For example, the highest concentration of alerts appears near latitude ~41.85 and longitude ~-87.65, likely corresponding to a central or downtown Chicago area. Another significant cluster is located near latitude ~41.72 and longitude ~-87.70, likely in the southern region of Chicago.

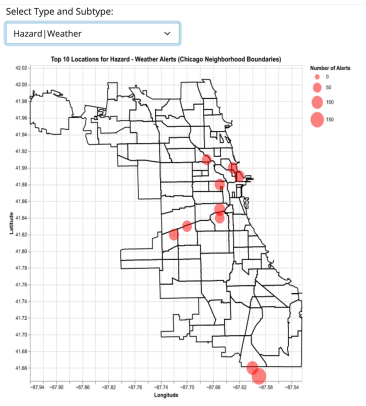


Figure 4: Hazard Weather Alerts

d.

Question: Were are alerts for hazard due to weather most common?

Answer: The most common alerts for weather hazards are concentrated in: - Central Chicago around latitude 41.86, longitude -87.62. - Southern neighborhoods around latitude 41.72, longitude -87.70. - Other significant alerts can be seen along the lakefront area near downtown.

e. To improve the dashboard's utility, adding a column for "Time of Day" (could be something like morning, afternoon, evening) could enable users to explore temporal traffic patterns, such as identifying high-alert locations during rush hours or late-night events. It would allow the dashboard to answer time-specific questions, such as: "Where are the most alerts during morning hours?", "Which areas experience the most congestion at night?"

App #2: Top Location by Alert Type and Hour Dashboard (20 points)

1.

a.

Collapsing the dataset by the ts column is not ideal because timestamps (ts) are unique and very granular. Aggregating by ts would lead to a dataset with almost the same number of rows as the original dataset, which wouldn't provide useful insights or reduce complexity.

Instead, extracting the hour from the timestamp makes more sense. This lets you group by hour (hour), type, and subtype, which creates a meaningful level of aggregation for analysis.

b.

```
# Ensure the `ts` column is in datetime format
merged_df['ts'] = pd.to_datetime(merged_df['ts'])

# Extract the hour from the timestamp and format it as HH:00
merged_df['hour'] = merged_df['ts'].dt.strftime('%H:00')

# Create the collapsed dataframe
collapsed_df = merged_df.groupby(
    ['hour', 'updated_type', 'updated_subtype', 'binned_latitude',
↪  'binned_longitude']
).size().reset_index(name='alert_counts')

print(f"There are {len(collapsed_df)} rows of this Dataframe.")

# Save the collapsed dataframe
collapsed_df.to_csv("./top_alerts_map_byhour/alert_counts.csv")
```

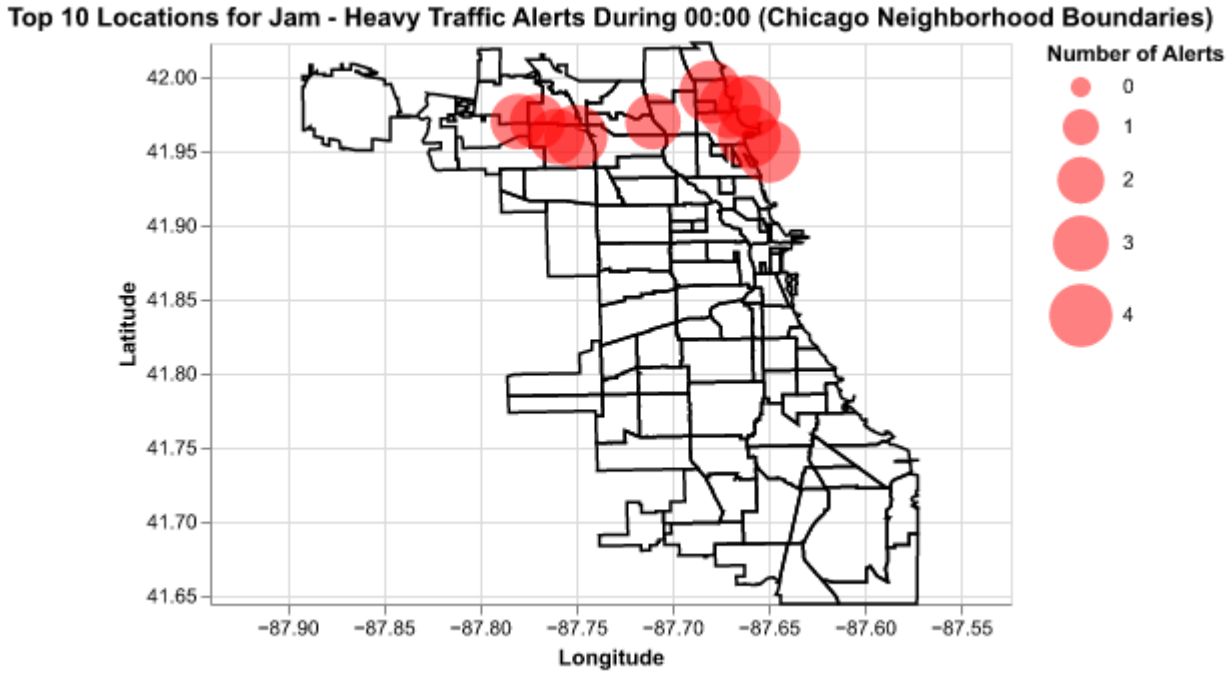
```
There are 70495 rows of this Dataframe.
```

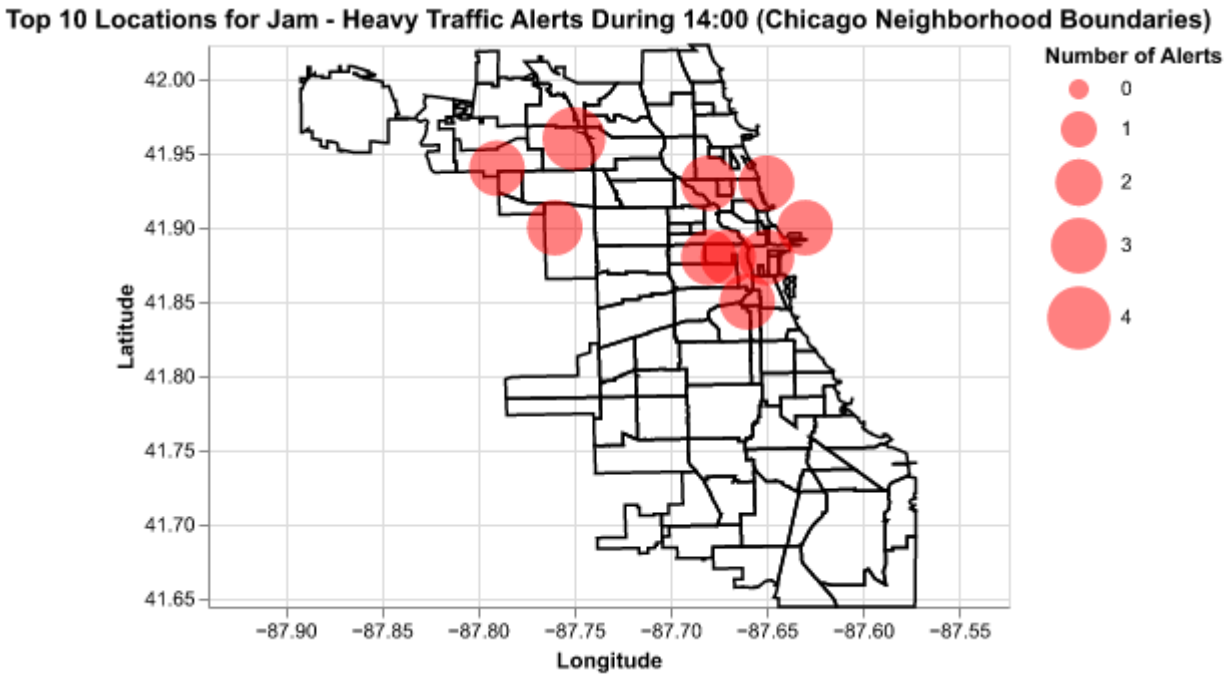
c.

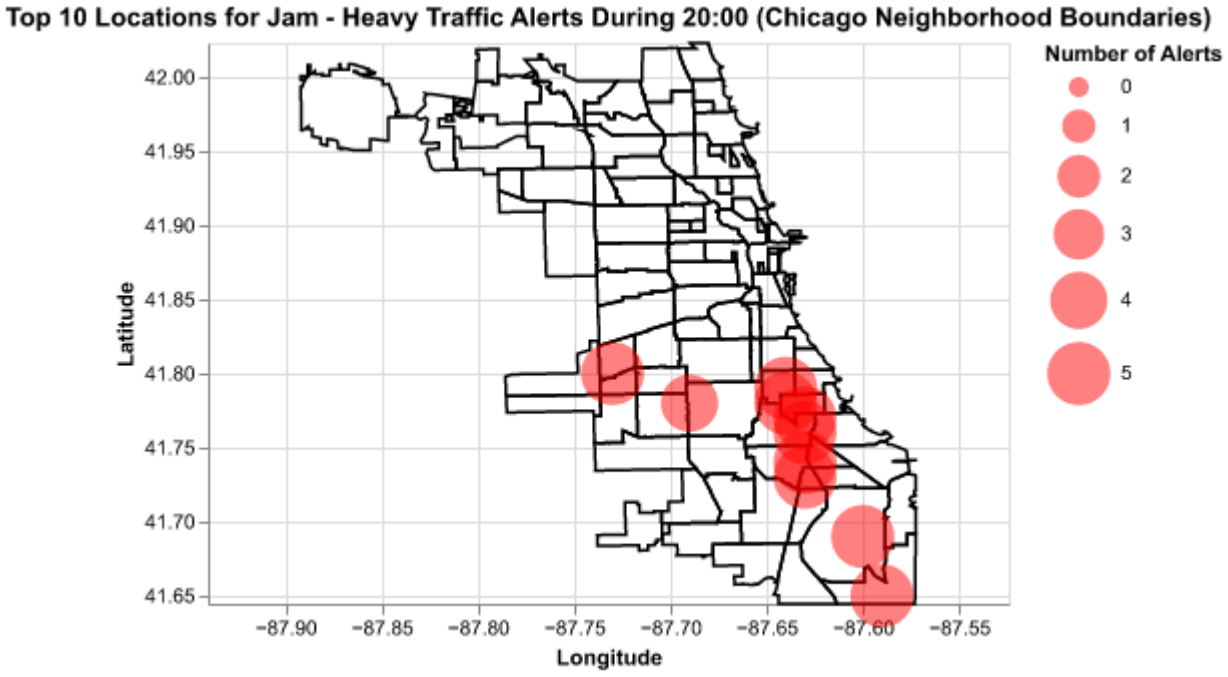
```
# Load data
file_path = "./top_alerts_map_byhour/alert_counts.csv"
df = pd.read_csv(file_path)

# Filter for 'Jam - Heavy Traffic' alerts
chosen_type = 'Jam'
chosen_subtype = 'Heavy Traffic'
filtered_df = df[
    (merged_df['updated_type'] == chosen_type) &
    (merged_df['updated_subtype'] == chosen_subtype)
]

# Define a function to create plots for a specific hour
def plot_top_alerts_by_hour(hour, base_map, longitude_limits,
↪  latitude_limits):
    # Filter data for the specified hour
    hour_filtered_df = filtered_df[filtered_df['hour'] == f"{hour:02}:00"]

    # Group by binned coordinates and count alerts, then get the top 10
```

```
    top_bins = (
        hour_filtered_df.groupby(['binned_latitude', 'binned_longitude'])
        .size()
        .reset_index(name='alert_counts')
        .sort_values(by='alert_counts', ascending=False)
        .head(10)
    )

    # Scatter plot for the top 10 locations
    scatter_plot = alt.Chart(top_bins).mark_circle(
        color='red',
        opacity=0.5
    ).encode(
        x=alt.X('binned_longitude:Q',
                title='Longitude',
                scale=alt.Scale(domain=longitude_limits)),
        y=alt.Y('binned_latitude:Q',
                title='Latitude',
                scale=alt.Scale(domain=latitude_limits)),
        size=alt.Size('alert_counts:Q',
                      title='Number of Alerts',
                      scale=alt.Scale(range=[100, 1000])),
        tooltip=[
            alt.Tooltip('binned_latitude:Q', title='Latitude'),
            alt.Tooltip('binned_longitude:Q', title='Longitude'),
            alt.Tooltip('alert_counts:Q', title='Number of Alerts')
        ]
    ).properties(
    width=400,
    height=280
  )

    # Create the layered map
    layered_map = (base_map + scatter_plot).properties(
    title=f"Top 10 Locations for {chosen_type} - {chosen_subtype} Alerts
 ↪  During {hour:02}:00 (Chicago Neighborhood Boundaries)"
  )

    # Combine the base map and scatter plot
    return layered_map

# Generate plots for three different hours (use the same longitude and
 ↪  latitude ranges.)
```

```
hour_plots = [plot_top_alerts_by_hour(
  hour, base_map,longitude_limits, latitude_limits) for hour in [0, 14, 20]
]

# Display the plots (separately)
for plot in hour_plots:
    plot.display()
```




Top 10 Locations for Jam - Heavy Traffic Alerts During 14:00 (Chicago Neighborhood Boundaries)
Number of Alerts
0
1
2
3
4
Latitude
42.00
41.95
41.90
41.85
41.80
41.75
41.70
41.65
−87.90
−87.85
−87.80
−87.75
−87.70
−87.65
−87.60
−87.55
Longitude


Top 10 Locations for Jam - Heavy Traffic Alerts During 20:00 (Chicago Neighborhood Boundaries)
Number of Alerts
0
1
2
3
4
5
Latitude
42.00
41.95
41.90
41.85
41.80
41.75
41.70
41.65
−87.90
−87.85
−87.80
−87.75
−87.70
−87.65
−87.60
−87.55
Longitude

2.

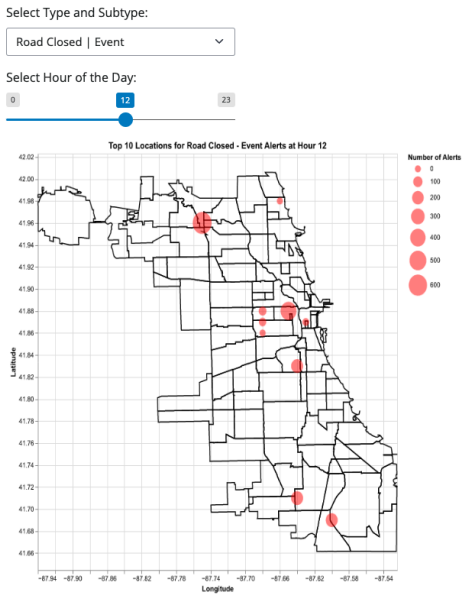
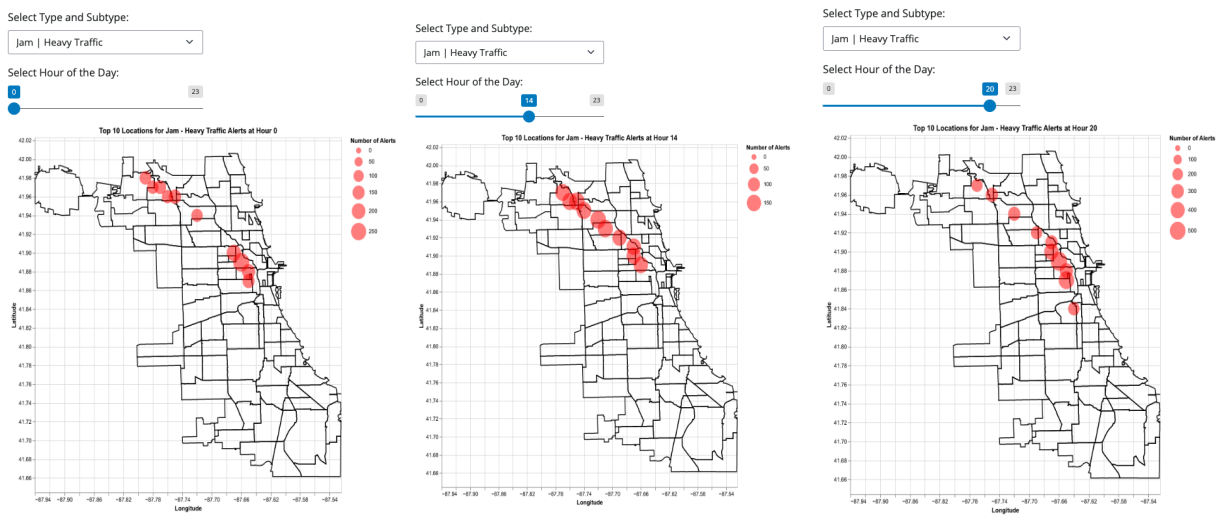


Figure 5: Dropdown Menu (by Hour)

a.



Jam Heavy Traffic at 0:00	Jam Heavy Traffic at 14:00	Jam Heavy Traffic at 20:00

b.

c. From the dashboard, it seems that road construction is done more during night hours.

App #3: Top Location by Alert Type and Hour Dashboard (20 points)

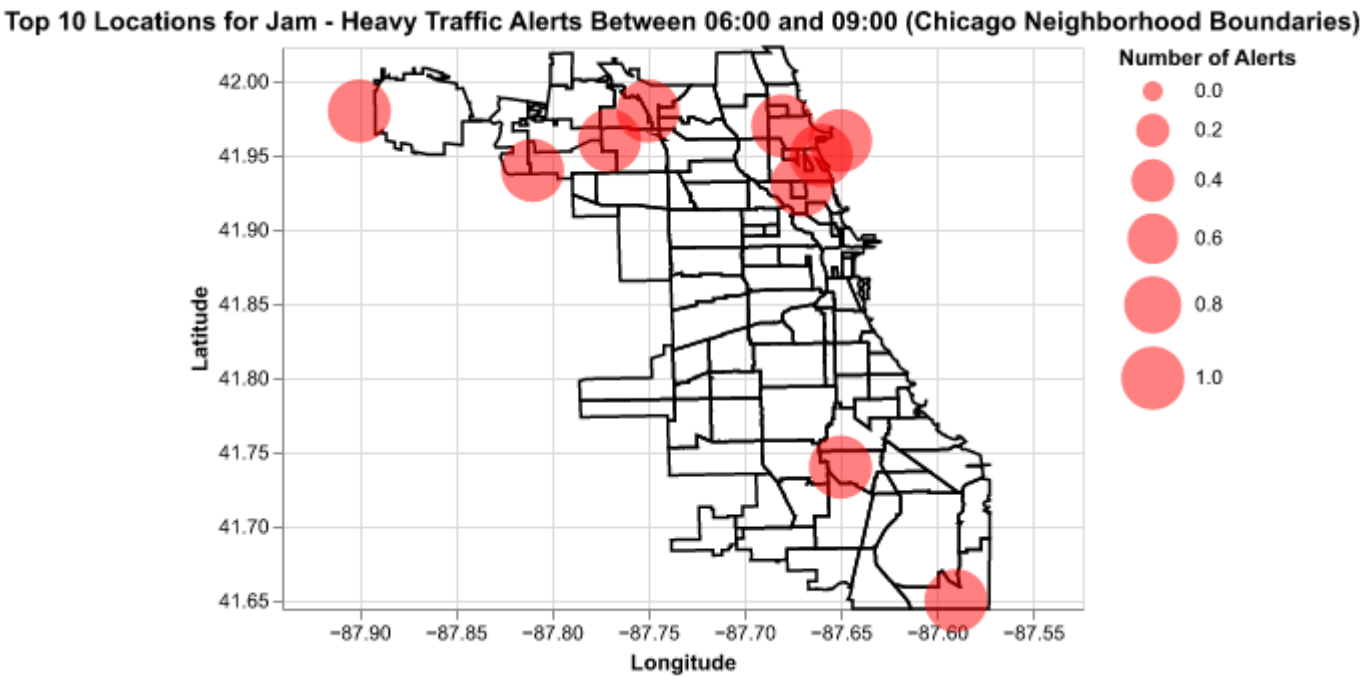
1.

a. Collapsing the dataset by a range of hours would not be the best idea for this scenario. Collapsing the data into predefined hour ranges (e.g., 6AM-9AM) will make the app less flexible. Users won't be able to dynamically select custom hour ranges (e.g., 7AM-11AM), which defeats the purpose of adding the slider range functionality. Instead of pre-collapsing by range, it's better to collapse by individual hours and aggregate dynamically based on the user-selected range. This reduces the risk of overloading the app while keeping the data flexible.

b.

```
# Load data
file_path = "./top_alerts_map_byhour/alert_counts.csv"
df = pd.read_csv(file_path)

# Filter for 'Jam - Heavy Traffic' alerts between 6AM and 9AM
chosen_type = 'Jam'
chosen_subtype = 'Heavy Traffic'
hour_lower_bound = "06"
hour_upper_bound = "09"
filtered_df = df[
    (merged_df['updated_type'] == chosen_type) &
    (merged_df['updated_subtype'] == chosen_subtype) &
    (merged_df['hour'] >= f"{hour_lower_bound}:00") & (merged_df['hour'] <
 ↪  f"{hour_upper_bound}:00")
]

# Group by binned coordinates and count alerts, then get the top 10
top_bins = (
    filtered_df.groupby(['binned_latitude', 'binned_longitude'])
    .size()
    .reset_index(name='alert_counts')
    .sort_values(by='alert_counts', ascending=False)
    .head(10)
)

# Scatter plot for the top 10 locations
scatter_plot = alt.Chart(top_bins).mark_circle(
```

```
    color='red',
    opacity=0.5
).encode(
    x=alt.X('binned_longitude:Q',
            title='Longitude',
            scale=alt.Scale(domain=longitude_limits)),
    y=alt.Y('binned_latitude:Q',
            title='Latitude',
            scale=alt.Scale(domain=latitude_limits)),
    size=alt.Size('alert_counts:Q',
                  title='Number of Alerts',
                  scale=alt.Scale(range=[100, 1000])),
    tooltip=[
        alt.Tooltip('binned_latitude:Q', title='Latitude'),
        alt.Tooltip('binned_longitude:Q', title='Longitude'),
        alt.Tooltip('alert_counts:Q', title='Number of Alerts')
    ]
).properties(
    width=400,
    height=280
)

# Create the layered map
layered_map = (base_map + scatter_plot).properties(
title=f"Top 10 Locations for {chosen_type} - {chosen_subtype} Alerts Between
 ↪  {hour_lower_bound}:00 and {hour_upper_bound}:00 (Chicago Neighborhood
 ↪  Boundaries)"
)

layered_map.display()
```



2.

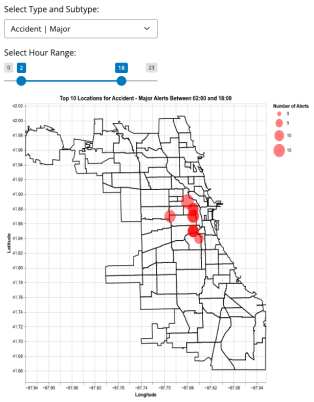


Figure 6: Dropdown Menu (by Hour Range)

a.

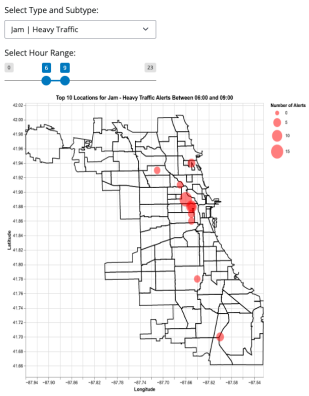


Figure 7: Jam Heavy Traffic Between 06:00 and 09:00

b.

3.

a. The possible values of input.switch_button are: True (when toggled to range of hours) and False (when toggled to a single hour).

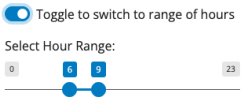
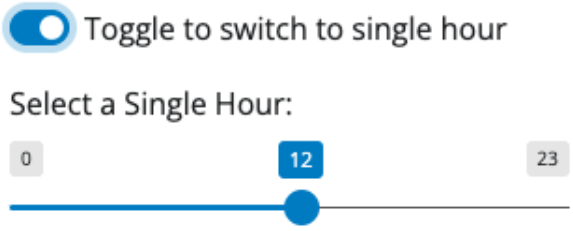
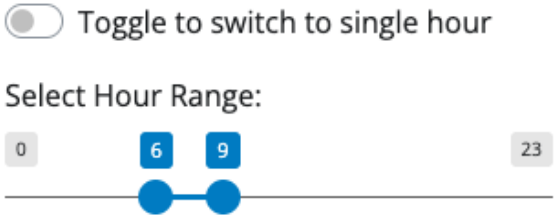
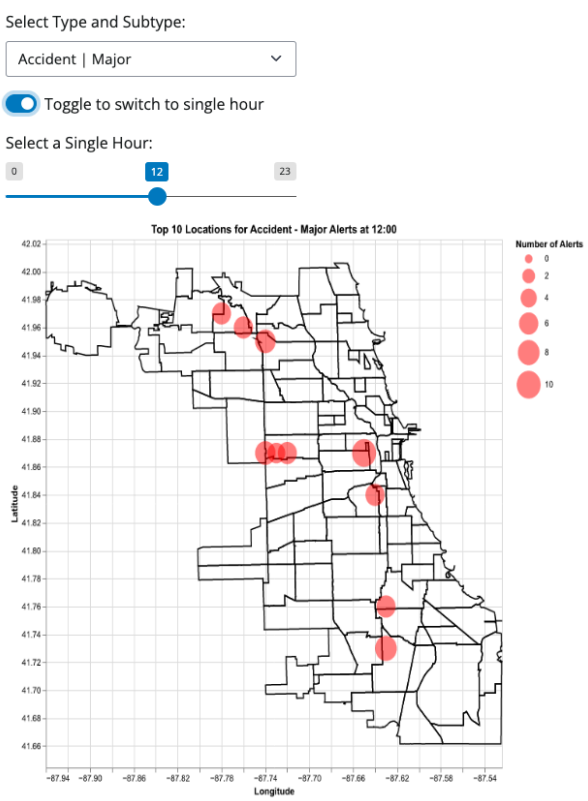


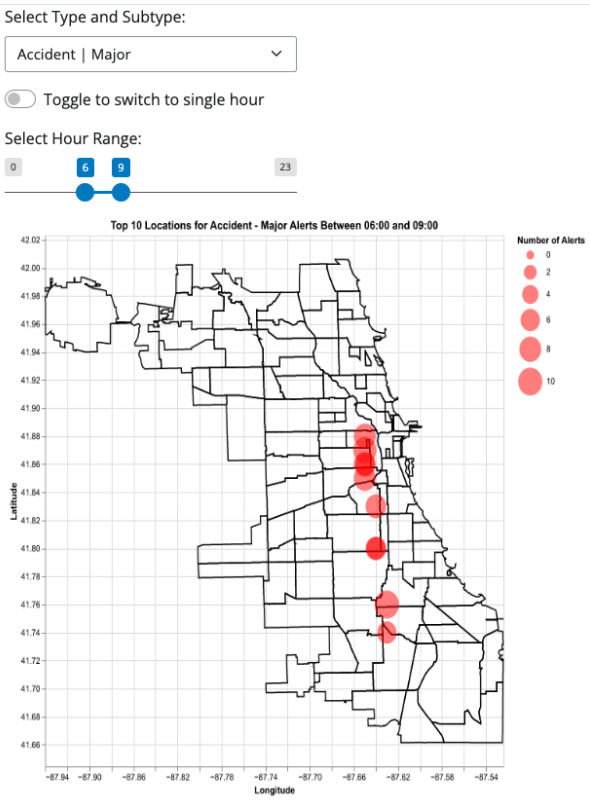
Figure 8: Dropdown Menu with Switch Button

	
When the switch button is toggled, the slider for a single hour is shown	When the switch button is not toggled, the slider for a range of hours is shown

b.



Plot Generated with the slider for a single hour



Plot Generated with the slider for a range of hours

c.

d. To achieve a plot similar to the one below, I need to add a time period classification (e.g., introducing a new categorical variable that classifies hours into broader categories like “Morning”, “Afternoon”, and “Evening”). In addition, I need to add an encoding for color based on the time_period variable to visually differentiate between periods (e.g., red for Morning and blue for Afternoon). I also need to add an encoding for stroke or fill to make it visually consistent with the plot below. Finally, I need to update the UI and filtering logic. Specifically, the app should allow the user to filter data based on the selected time period in addition to the type, subtype, and hour range.